Software Engineering Institute

As-If Infinitely Ranged Integer Model, Second Edition

Roger Dannenberg (School of Computer Science, Carnegie Mellon University)
Will Dormann (Software Engineering Institute)
David Keaton (Software Engineering Institute)
Thomas Plum (Plum Hall, Inc.)
Robert C. Seacord (Software Engineering Institute)
David Svoboda (Software Engineering Institute)
Alex Volkovitsky (Software Engineering Institute)
Timothy Wilson (Software Engineering Institute)

April 2010

TECHNICAL NOTE
CMU/SEI-2010-TN-008

CERT Program

Unlimited distribution subject to the copyright.

http://www.sei.cmu.edu

Carnegie Mellon

Note: This report is the second edition of CMU/SEI-2009-TN-023, a technical note published by the Software Engineering Institute in 2009.

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Copyright 2010 Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

External use. This document may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other external and/or commercial use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about SEI publications, please visit the library on the SEI website (http://www.sei.cmu.edu/library).

Table of Contents

Acknowledgments **vii**

Abstract **ix**

1 Integral Security **1**
- 1.1 Signed Integer Overflow 1
- 1.2 Unsigned Integer Wrapping 2
- 1.3 Conversion Errors 2

2 AIR Integer Model **3**
- 2.1 Implementation Methods 4
- 2.2 Undefined Behavior 5
  - 2.2.1 Multiplicative Operators 6
  - 2.2.2 Shifts 6
  - 2.2.3 Fussy Overflows 7
- 2.3 Enabling and Disabling Unsigned Integer Wrapping 8
- 2.4 Integer Promotions and the Usual Arithmetic Conversions 8
- 2.5 Integer Constants 9
- 2.6 Expressions Involving Integer Variables and Constants 9
- 2.7 Runtime-Constraint Handling 9
- 2.8 Optimizations 10
- 2.9 The `rsize_t` Type 11
- 2.10 Pointer Arithmetic 11

3 Related Work **12**
- 3.1 The GCC `-ftrapv` Flag 12
- 3.2 Precondition Testing 12
- 3.3 Saturation Semantics 14
- 3.4 Overflow Detection 15
- 3.5 Runtime Integer Checking (RICH) 16
- 3.6 Clang Implementation 16
- 3.7 GCC no-undefined-overflow 17
- 3.8 Testing Methods 17

4 Performance & Efficacy Study **18**
- 4.1 Performance Study 18
- 4.2 Efficacy Study 19

5 Conclusions **26**

Bibliography **27**

List of Figures

Figure 1: JasPer Defects 20

Figure 2: FFmpeg Defects 21

Figure 3: Combined Defects 21

List of Tables

Table 1: Critical Undefined Behavior 4

Table 2: Saturation Semantics 14

Table 3: SPECINT2006 Macro-Benchmark Runs 19

Acknowledgments

We would like to acknowledge the contributions of the following individuals to the research presented in this paper: Alex Volkovitsky, Chad Dougherty, Ian Lance Taylor, Richard Guenther, David Chisnall, Tzi-cker Chiueh, Huijia Lin, Joseph Myers, Raunak Rungta, Alexey Smirnov, Rob Johnson, and David Brumley. We would also like to acknowledge the contribution of our technical editors and librarians: Pennie Walters, Edward Desautels, Alexa Huth, and Sheila Rosenthal. This research was supported by the U.S. Department of Defense (DoD), the U.S. Department of Homeland Security (DHS) National Cyber Security Division (NCSD), and CyLab at Carnegie Mellon under grants DAAD19-02-1-0389 from the Army Research Office.

Abstract

Integers represent a growing and underestimated source of vulnerabilities in C and C++ programs. This report presents the as-if infinitely ranged (AIR) integer model that provides a largely automated mechanism for eliminating integer overflow and truncation and other integral exceptional conditions. The AIR integer model either produces a value equivalent to that obtained using infinitely ranged integers or results in a runtime-constraint violation. Instrumented fuzz testing of libraries that have been compiled using a prototype AIR integer compiler has been effective in discovering vulnerabilities in software with low false positive and false negative rates. Furthermore, the runtime overhead of the AIR integer model is low enough for typical applications to enable it in deployed systems for additional runtime protection.

1 Integral Security

The majority of software vulnerabilities result from coding errors. For example, 64% of the vulnerabilities in the National Vulnerability Database in 2004 resulted from programming errors [Heffley 2004]. The C and C++ languages are particularly prone to vulnerabilities because of the lack of type safety in these languages [Seacord 2005].

In 2007, MITRE reported that buffer overflows remain the number one issue as reported in operating system (OS) vendor advisories. It also reported that integer overflow, barely in the top 10 overall in the years preceding the report, was number two in OS vendor advisories [Christy 2007].

Integer errors and vulnerabilities occur when programmers reason about infinitely ranged mathematical integers, while implementing their designs with the finite precision, integral data types supported by hardware and language implementations.

Integer values that originate from untrusted sources and are used in the following ways can easily result in vulnerabilities:

- as an array index in pointer arithmetic
- as a length or size of an object
- as the bound of an array (e.g., a loop counter)
- as an argument to a memory-allocation function

The following sections describe integer behaviors that have resulted in real-world vulnerabilities.

1.1 Signed Integer Overflow

Signed integer overflow is undefined behavior in C, allowing implementations to silently wrap (the most common behavior), trap, or both. Because signed integer overflow produces a silent wraparound in most existing C and C++ implementations, some programmers assume that this is a well-defined behavior.

Conforming C and C++ compilers can deal with undefined behavior in many ways, such as ignoring the situation completely (with unpredictable results), translating or executing the program in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), or terminating a translation or execution (with the issuance of a diagnostic message). Because compilers are not obligated to generate code for undefined behaviors, those behaviors are candidates for optimization. By assuming that undefined behaviors will not occur, compilers can generate code with better performance characteristics. For example, GCC Version 4.1.1 optimizes out integer expressions that depend on undefined behavior for all optimization levels.

Signed integer overflow is frequently not considered to be a problem for hardware that detects it, because overflow is undefined behavior.

1.2 Unsigned Integer Wrapping

Although unsigned integer wrapping is well defined by the C standard as having modulo behavior, unexpected wrapping has led to numerous software vulnerabilities. A real-world example of vulnerabilities resulting from unsigned integer wrapping occurs in memory allocation. Wrapping can occur in `calloc()` and other memory allocation functions when the size of a memory region is being computed.[1] As a result, the buffer returned is smaller than the requested size, and that can lead to a subsequent buffer overflow.

For example, the following code fragments may lead to wrapping vulnerabilities, where `count` is an unsigned integer:

```
C: p = calloc(sizeof(element_t), count);
C++: p = new ElementType[count];
```

The wrapping of calculations internal to these functions may result in too little storage being allocated and subsequent buffer overflows. However, the draft standard titled *Programming Languages—C++, Final Committee Draft* requires that a new expression throw an instance of `std::bad_array_new_length` on integer overflow [ISO 2010].

Another well-known vulnerability resulting from unsigned integer wrapping occurred in the handling of the comment field in JPEG files [Solar Designer 2000].

1.3 Conversion Errors

Integer conversions, both implicit and explicit (using a cast), must be guaranteed not to result in lost or misinterpreted data [Seacord 2008].

The only integer type conversions that are guaranteed to be safe for all data values and all possible conforming implementations are conversions of an integral value to a wider type of the same signedness. Conversion of an integer to a smaller type results in truncation of the high-order bits.

Consequently, conversions from an integer with greater precision to an integer type with lesser precision can result in truncation, if the resulting value cannot be represented in the smaller type. Conversions to an integer of the same precision but different signedness can lead to misinterpreted data.

---

[1] http://www.securityfocus.com/bid/5398

2 AIR Integer Model

The purpose of the AIR integer model is to produce either a value that is equivalent to that obtained using infinitely ranged integers or a runtime-constraint violation. The model applies to both signed and unsigned integers, although either may be enabled or disabled per compilation unit using compiler options.

Implementations must declare that they are implementing the AIR integer model with a predefine, `__STDC_ANALYZABLE__`. The term *analyzable* is used here to indicate that the resulting system is easier to analyze because undefined behaviors have been defined and because the analyzer (either a tool or human) can safely assume that integer operations will result in an as-if infinitely ranged value or trap.

Traps are implemented using the existing hardware traps (such as divide-by-zero) or by invoking a runtime-constraint handler. Whether a program traps for given inputs depends on the exact optimizations carried out by a particular compiler version. If required, a programmer can implement a custom runtime-constraint handler to set a flag and continue (using the indeterminate value that was produced). In the future, an implementation that also supports C++ might throw an exception or invoke `terminate()`, rather than invoke a runtime-constraint handler. Alternatively, the runtime-constraint handler can throw an exception. We have not attempted to evaluate these, or other, alternatives for C++.

A trap representation is a set of bits that, when interpreted as a value of a specific type, causes undefined behavior. Trap representations are most commonly seen on floating point and pointer values. However, in theory, almost any type could have trap representations.

An *observation point* occurs at an output, including a volatile object access. AIR integers do not *require* a trap for every integer overflow or truncation error. In the AIR integer model, it is acceptable to delay catching an incorrectly represented value until an observation point is reached or just before it causes *a critical undefined behavior* [Jones 2009]. The trap may occur any time between the overflow or truncation and the output or critical undefined behavior. This model improves the ability of compilers to optimize, without sacrificing safety and security.

Critical undefined behavior is a means of differentiating between behaviors that can perform an out-of-bounds store and those that cannot. An out-of-bounds store is defined in the C1X draft standard titled *Programming Languages—C, Committee Draft* as an (attempted) access that, at runtime and for a given computational state, would modify (or, for an object declared volatile, fetch) one or more bytes that lie outside the bounds permitted by the C1X draft standard [Jones 2009].

The critical undefined behaviors (with references to the section in the C1X draft in which they are defined) are shown in Table 1.

Table 1: Critical Undefined Behavior

C1X Section	Critical Undefined Behavior
6.2.4	An object is referred to outside of its lifetime.
6.3.2.1	An lvalue does not designate an object when evaluated.
6.3.2.3	A pointer is used to call a function whose type is not compatible with the pointed-to type.
6.5.3.2	The operand of the unary `*` operator has an invalid value.
6.5.6	Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary `*` operator that is evaluated.
7.1.4	An argument to a library function has an invalid value or a type not expected by a function with a variable number of arguments.
7.21.3	The value of a pointer that refers to space deallocated by a call to the `free` or `realloc` function is used.
7.22.1, 7.27.4	A string or wide-string utility function is instructed to access an array beyond the end of an object.

In the AIR integer model, when an observation point is reached and before any critical undefined behavior occurs, any integer value in the output is correctly represented ("as-if infinitely ranged") provided that traps have not been disabled and no traps have been raised. Optimizations are encouraged, provided the model is not violated.

2.1 Implementation Methods

The AIR integer method permits a wide range of implementation methods, some of which might apply to different environments and implementations:

- Overflow or truncation can set a flag that compiler-generated code will test later.
- Overflow or truncation can immediately invoke a runtime-constraint handler.
- The testing of flags can be performed at an early point (such as within the same full expression) or delayed (subject to some restrictions).

For example, in the following code

```
i = k + 1;
j = i * 3;
if (m < 0)
  a[i] = . . .;
```

the variable `j` does not need to be checked within this code fragment (but may need to be checked later). The variable `i` does not need to be checked unless and until the `a[i]` expression is evaluated.

Compilers may choose a single, cumulative integer exception flag in some cases and one flag per variable in others, depending on what is most efficient in terms of speed and storage for the particular expressions involved. For example, in the following code

```
  x++;
  y++;
  z++;
  printf("%d", x);
```

the call to `printf()` is an observation point for the variable `x`. Any of the operations `x++`, `y++`, or `z++` can result in an overflow. Consequently, it is necessary to test the value of the exception flag prior to the observation point (the call to `printf()`) and invoke the runtime-constraint handler if the exception flag is set:

```
// compiler clears integer exception flags
  x++;
  y++;
  z++;
  if (/* integer exception flags are set */)
    runtime_constraint_handler();
  printf("%d", x);
```

If only a single exception flag is used, one or more of the variables may contain an incorrectly represented value, but we cannot know which one. Consequently, the runtime-constraint handler will be invoked if any of the increment operations resulted in an overflow. In that case, it may be preferable for the compiler to generate a separate exception flag for `x` so that the runtime-constraint handler need only be invoked if `x++` overflows.

Portably, if the code reaches an observation point without invoking a runtime-constraint handler, a programmer can only assume that all observable integer values are represented correctly. If a runtime-constraint error occurs, all integer values that have been modified since the last observation point contain indeterminate values. In cases where the programmer wants to rerun the calculation using a higher or arbitrary-precision integer, the programmer would need to recalculate the values for all indeterminate values.

Ideally, while we would like to eliminate implementation-defined behavior in the AIR integer model, sufficient latitude must be provided for compiler implementers to optimize the resulting executable.

2.2 Undefined Behavior

One of the goals of the AIR integer model is to eliminate previously undefined behaviors by providing optional predictable semantics for areas of C that are presently undefined (at some optimization cost). Changes from the existing, unbounded, undefined behavior that pose serious implementation problems in practice were not adopted under the model.

The following cases receive special handling in the AIR integer model.

2.2.1 Multiplicative Operators

There is no defined infinite-precision result for division by zero. Processors typically trap, but that may not be universal. The AIR integer model requires trapping.

When integers are divided, the result of the / operator is the algebraic quotient with any fractional part discarded. If the quotient `a/b` is representable, the expression `(a/b)*b + a%b` is equal to `a`. Otherwise, the behavior of both `a/b` and `a%b` is undefined by C99, but processors commonly trap. For example, when using the IA-32 `idiv` instruction, dividing `INT_MIN` by `-1` results in a division error and generates an interrupt on vector 0 because the signed result (quotient) is too large for the destination [Intel 2004]. The AIR integer model requires trapping if the quotient is not representable.

The ISO/IEC JTC1/SC22/WG14 C standards committee discussed the behavior of `INT_MIN % -1` on the WG14 reflector and at the April 2009 Markham meeting [Hedquist 2009]. The committee agreed that, mathematically, `INT_MIN % -1` equals 0. However, instead of producing the mathematically correct result, some architectures may trap. For example, implementations targeting the IA-32 architecture use the `idiv` instruction to determine the remainder. Consequently, `INT_MIN % -1` results in a division error and generates an interrupt on vector 0.

At the same Markham meeting, some committee members argued that C99 requires a C program computing `INT_MIN % -1` to produce `0` because `0` is representable. Others argued that C99 left the computation as undefined because `INT_MIN / -1` is not representable. The committee decided that requiring C programs to produce `0` would render some compilers noncompliant with the standard and that adding this corner case could add a significant overhead. Consequently, the C1X draft standard [Jones 2009] has been amended to state explicitly that if `a/b` is not representable, `a%b` is undefined.

The AIR integer model requires that `a % -1` equals `0` for all values of `a` or, alternatively, trapping is performed. This violates the literal interpretation of "as-if infinite range" but reflects a concession to practical implementation issues.

By comparison, in Java, an integer division or integer remainder operator throws an `ArithmeticException` if the value of the right-hand operand expression is zero [Gosling 2005]. The remainder operation for operands that are integers after binary numeric promotion produces a result value such that `(a/b)*b+(a%b)` is equal to `a`. This identity holds even when the dividend is the negative integer of the largest possible magnitude for its type and the divisor is -1 (the remainder is 0).

2.2.2 Shifts

Shifting by a negative number of bits or by more bits than exist in the operand is undefined behavior in C99 and, in almost every case, indicates a bug (logic error). Signed left shifts of negative values or cases where the result of the operation is not representable in the type are undefined in C99 and implementation-defined in C90. Processors may reduce the shift amount modulo some quantity larger than the width of the type. For example, 32-bit shifts are implemented using the following instructions on IA-32:

```
sa[rl]l   %cl, %eax
```

The `sa[rl]l` instructions take a bit mask of the least significant 5 bits from `%cl` to produce a value in the range [0, 31] and then shift `%eax` that many bits.

64-bit shifts become

```
sh[rl]dl  %eax, %edx
sa[rl]l   %cl, %eax
```

where `%eax` stores the least significant bits in the double word to be shifted and `%edx` stores the most significant bits.

In the AIR integer model, shifts by negative amounts or amounts outside the width of the type trap because the results are not representable without overflow; consistent with rule INT34-C of the CERT C Secure Coding Standard: “Do not shift a negative number of bits or more bits than exist in the operand” [Seacord 2008].

In the AIR integer model, signed left shifts on negative values must not trap if the result is representable without overflow. If the value is not representable in the type, the implementation must trap. For example, `a << b == a * 2`^{b} if `b >= 0` and `a * 2`^{b} is representable without overflow in the type. For right shifts, a >> b == a / 2^{b} if `a >= 0, b >= 0`, and 2^{b} is representable without overflow in the type. If `a < 0, b >= 0`, and 2^{b} is representable without overflow in the type, `a >> b == -1 + (a + 1) / 2`^{b}. Unsigned left shifts never trap under the AIR integer model because unsigned left shifts are generally perceived by programmers as losing data, and a large amount of existing code assumes modulo behavior. For example, in the following code from the JasPer image processing library,[2] Version 1.900.1, `tmpval` has `uint_fast32_t` type:

```
while (--n >= 0) {
  c = (tmpval >> 24) & 0xff;
  if (jas_stream_putc(out, c) == EOF) {
    return -1;
  }
  tmpval = (tmpval << 8) & 0xffffffff;
}
```

The modulo behavior of `tmpval` is assumed in the left shift operation.

2.2.3 Fussy Overflows

One problem with trapping is *fussy overflows*, which are overflows in intermediate computations that do not affect the resulting value. For example, on two’s complement architectures, the following code

```
int x = /* nondeterministic value */;
x = x + 100 - 1000;
```

overflows for values of `x > INT_MAX - 100` but underflows during the subsequent subtraction, resulting in a correct as-if infinitely ranged integer value.

---

[2] http://www.ece.uvic.ca/~mdadams/jasper/

In this case, most compilers will perform constant folding to simplify the above expression to `x - 900`, eliminating the possibility of a fussy overflow. However, there are situations where this will not be possible, for example:

```
int x = /* nondeterministic value */;
int y = /* nondeterministic value */;
x = x + 100 - y;
```

Because this expression cannot be optimized, a fussy overflow may result in a trap, and a potentially successful operation may be converted into an error condition.

2.3 Enabling and Disabling Unsigned Integer Wrapping

The default behavior under the AIR integer model is to trap unsigned integer wrapping.

Unsigned integer semantics are problematic because unsigned integer wrapping poses a significant security risk but is well defined by the C standard. Also, in legacy code, the wrapping behavior can be critical to correct behavior. Consequently, it is necessary to provide mechanisms to enable and disable wrapping for unsigned integers.

It is theoretically possible to introduce new identifiers, such as `__wrap` and `__trap`, to be used as named attributes to enable or disable wrapping for individual integer variables, both signed and unsigned. They could be implemented as variable attributes in GCC or using `__declspec` or a similar mechanism in Microsoft Visual Studio. Enabling or disabling wrapping and trapping per variable has implications for the type system: for example, what happens when you combine a wrapping variable with a trapping variable? It also has implications for type safety: for example, what happens when you pass a trapping variable as an argument to a function that accepts a wrapping parameter?

Because of these added complications, the AIR integer model only supports enabling or disabling unsigned integer wrapping per compilation unit.

Compiler options can be provided to enable or disable wrapping for all unsigned integer variables per compilation unit. Existing code that depends on modulo behavior for unsigned integers should be isolated in a separate compilation unit and compiled with wrapping disabled.

When an unsigned integer defined in one compilation unit compiled with wrapping semantics is combined with another unsigned integer defined in a separate compilation unit with trapping semantics, the resulting value has the default behavior of the compilation unit in which the operation occurs.

Because a large number of exploitable software vulnerabilities result from unsigned integer wrapping, we strongly recommend that the trap behavior be the default for all new code and, for as much legacy code as possible, consistent with adequate testing and code review.

2.4 Integer Promotions and the Usual Arithmetic Conversions

In cases where a compilation unit is compiled with wrapping disabled for unsigned integers, it is possible that operations can take place between signed integers with trapping semantics and unsigned integers with wrapping semantics. In these cases, the semantics of the resulting variable (trapping or wrapping) depends on the integer promotions and the usual arithmetic conversions defined by C99. In cases where the resulting variable is a signed integer type, trapping semantics apply; in cases where the resulting value is an unsigned integer type, wrapping semantics are used.

2.5 Integer Constants

In C99, it is a constraint violation if the value of a constant is outside the range of representable values for its type. A C99-conforming implementation must produce at least one diagnostic message (identified in an implementation-defined manner) if a preprocessing translation unit or other translation unit contains a violation of any constraint.

For constant expressions, the AIR integer model requires that the compiler must use arbitrary-precision signed arithmetic to evaluate an integer constant expression (even an unsigned one) and then issue a fatal diagnostic if the final result does not fit the appropriate type.

For example, the expression

```
((unsigned)0 - 1)
```

produces a constraint violation and should result in a fatal diagnostic if compiled.

2.6 Expressions Involving Integer Variables and Constants

Because of macro expansion, another common case in C programs involve expressions that include some number of variables and some number of constant values such as

```
V1 + 1u + V2 - 2u
```

In this case, the compiler can reorder the expressions and reduce to a single constant value, for example

```
V1 + V2 - 1u
```

regardless of whether it is compiled with trapping enabled or disabled for unsigned integer values.

2.7 Runtime-Constraint Handling

Most functions defined by ISO/IEC TR 24731-1 [ISO 2006] and by the bounds-checking interfaces annex of the C1X draft standard [Jones 2009] include as part of their specification a list of runtime constraints, violations of which can be consistently handled at runtime. Library implementations must verify that the runtime constraints for a function are not violated by the program. If a runtime constraint is violated, the runtime-constraint handler currently registered with `set_constraint_handler_s()` is called.

Implementations are free to detect any case of undefined behavior and treat it as a runtime-constraint violation by calling the runtime-constraint handler. This license comes directly from the definition of undefined behavior. Consequently, the AIR implementation uses the runtime-constraint mechanisms defined by ISO/IEC TR 24731-1 and by the C1X draft standard for handling integer exceptional conditions.

2.8 Optimizations

An important consideration in adopting a new integer model is the effect on compiler optimization and vice versa. C-language experts are accustomed to evaluating the CPU cost of various proposals. A typical approach is to compare the CPU cost of solving the problem in the compiler versus the (zero) cost of not doing so. We submit that, for the AIR integer model, it would also be useful to consider the CPU cost of analyzable generated code versus the CPU cost of the programmer's extra program logic added to the intrinsic CPU cost of the optimized construct. This comparison justifies putting a greater burden on the compiler when compiling otherwise insecure constructs in analyzable mode. However, our current work uses the traditional approach to demonstrate that solving the problem does not introduce a large amount of overhead.

Regardless, performance is always an issue when evaluating new models, and it is important to preserve existing optimizations while discovering new ones. Consequently, the AIR integer model does not prohibit any optimizations that are permitted by the C standard but does require a diagnostic any time the compiler performs an optimization based on

- signed overflow wrapping
- unsigned wrapping
- signed overflow not occurring (although value-range analysis cannot guarantee it will not)
- unsigned wrapping not occurring (although value-range analysis cannot guarantee it will not)

For example, AIR integers allow optimizations based on algebraic simplification without a diagnostic:

```
(signed) (a * 10) / 10
```

This expression can be optimized to `a`. There is no need to preserve the possibility of trapping `a * 10`.

The expression

```
(a - 10) + (b - 10)
```

can be optimized to

```
(a + b) - 20
```

While there is a possibility that `(a + b)` will produce a trap, there is also a possibility that either `(a - 10)` or `(b - 10)` will result in a trap in the original expression. Provided that the application can be sure that each output is represented correctly, knowing whether a trap might have occurred by a different strategy does not really matter.

Optimizations that assume that integer overflow does not trap require a diagnostic because that assumption is inconsistent with the integer model. For example, certain optimizations operate on the basis that a loop must terminate by exactly reaching the limit `n`, and therefore the number of iterations can be determined by an exact division with no remainder such as

```
for (i = 0; i != n; i += 3)
```

This loop can be optimized to iterate for some number of terms determined by a code sequence that is only valid for exact division and not if `n/3` leaves a remainder. This loop should also be diagnosed because it violates rule MSC21-C of the CERT C Secure Coding Standard: “Use inequality to terminate a loop whose counter changes by more than one” [Seacord 2008].

Diagnostics are also required for optimizations on pointer arithmetic that assume wrapping cannot occur.

2.9 The `rsize_t` Type

The `rsize_t` type, defined as part of bounds-checked library functions [ISO 2006], can be used in a complementary fashion to AIR integers and is consequently subsumed as part of the overall solution. Functions that accept parameters of type `rsize_t` diagnose a constraint violation if the values of those parameters are greater than `RSIZE_MAX`. Extremely large object sizes are frequently a sign that an object’s size was calculated incorrectly. For example, negative numbers appear as very large positive numbers when converted to an unsigned type like `size_t`. For those reasons, it is sometimes beneficial to restrict the range of object sizes to detect errors. For machines with large address spaces, ISO/IEC TR 24731-1 recommends that `RSIZE_MAX` be defined as the smaller of these two, even if this limit is smaller than the size of some legitimate, but very large, objects:

- the size of the largest object supported
- `SIZE_MAX >> 1)`

The CERT C Secure Coding Standard recommends using `rsize_t` or `size_t` for all integer values representing the size of an object (rule INT01-C) [Seacord 2008].

2.10 Pointer Arithmetic

Pointer arithmetic is not part of the AIR integer model but can be checked by safe secure C/C++ (SSCC) methods [Plum 2005].

3 Related Work

This section describes existing and contemplated alternative approaches to the problem of integral security in C and explains why they don't adequately address the issues.

3.1 The GCC `-ftrapv` Flag

GCC provides an `-ftrapv` compiler option that provides limited support for detecting integer overflows at runtime. The GCC runtime system generates traps for signed overflow on addition, subtraction, and multiplication operations for programs compiled with the `-ftrapv` flag. This trapping is accomplished by invoking existing, portable library functions that test an operation's post-conditions and call the C library `abort()` function when results indicate that an integer error has occurred [Seacord 2005]. For example, the following function from the GCC runtime system is used to detect overflows resulting from the addition of signed 16-bit integers:

```
Wtype __addvsi3(Wtype a, Wtype b) {
   const Wtype w = a + b;
   if (b >= 0 ? w < a : w > a)
     abort ();
   return w;
}
```

The two operands are added, and the result is compared to the operands to determine whether an overflow condition has occurred. For `__addvsi3()`, if `b` is non-negative and `w < a`, an overflow has occurred and `abort()` is called. Similarly, `abort()` is called if `b` is negative and `w > a`.

The `-ftrapv` option is known to have substantial problems. The `__addvsi3()` function requires a function call and conditional branching, which can be expensive on modern hardware. An alternative implementation tests the processor overflow condition code, but it requires assembly code and is non-portable. Furthermore, the GCC `-ftrapv` flag only works for a limited subset of signed operations and always results in an `abort()` when a runtime overflow is detected. Discussions for how to trap signed integer overflows in a reliable and maintainable manner are ongoing within the GCC community.

3.2 Precondition Testing

Another approach to eliminating integer exceptional conditions is to test the values of the operands before an operation to prevent overflow and wrapping from occurring. This testing is especially important for signed integer overflow, which is undefined behavior and may result in a trap on some architectures (e.g., a division error on IA-32). The complexity of these tests varies significantly.

A precondition test for wrapping when adding two unsigned integers is relatively simple:

```
unsigned int ui1, ui2, usum;

/* Initialize ui1 and ui2 */

if (UINT_MAX - ui1 < ui2) {
  /* handle error condition */
}
else {
  usum = ui1 + ui2;
}
```

A strictly conforming test to ensure that a signed multiplication operation does not result in an overflow is significantly more involved:

```
signed int si1, si2, result;

/* Initialize si1 and si2 */

if (si1 > 0){
  if (si2 > 0) {
    if (si1 > (INT_MAX / si2)) {
      /* handle error condition */
    }
  }
  else {
    if (si2 < (INT_MIN / si1)) {
        /* handle error condition */
    }
  }
}
else {
  if (si2 > 0) {
    if (si1 < (INT_MIN / si2)) {
      /* handle error condition */
    }
  }
  else {
    if ((si1!=0) && (si2<(INT_MAX/si1))) {
      /* handle error condition */
    }
  }
}

result = si1 * si2;
```

Similar examples of precondition testing are shown in the CERT C Secure Coding Standard [Seacord 2008]:

- INT30-C. Ensure that unsigned integer operations do not wrap
- INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data
- INT32-C. Ensure that operations on signed integers do not result in overflow

Detecting an overflow in this manner can be relatively expensive, especially if the code is strictly conforming. Frequently, these checks must be in place before suspect system calls that may or may not perform similar checks prior to performing integral operations. Redundant testing by the caller and by the called is a style of defensive programming that has been largely discredited within the C and C++ community. The usual discipline in C and C++ is to require validation only on one side of each interface.

Furthermore, branches can be expensive on modern hardware, so programmers and implementers work hard to keep branches out of inner loops. This expense argues against requiring the application programmer to pretest all arithmetic values to prevent rare occurrences such as overflow. Preventing runtime overflow by program logic is sometimes easy, sometimes complicated, and sometimes extremely difficult. Clearly, some overflow occurrences can be diagnosed in advance by static-analysis methods. But no matter how good this analysis is, some code sequences still cannot be detected before runtime. In most cases, the resulting code is much less efficient than what a compiler could generate to detect overflow.

The underlying process of code generation may be immensely complicated. However, in general, it is best to avoid complexity in the code that end-user programmers are required to write.

3.3 Saturation Semantics

Verifiably in-range operations are often preferable to treating out-of-range values as an error condition because the handling of these errors has been shown to cause denial-of-service problems in actual applications (e.g., when a program aborts). The quintessential example of this incorrect handling is the failure of the Ariane 5 launcher, which resulted from an improperly handled conversion error that caused the processor to be shut down [Lions 1996].

A program that detects an imminent integer overflow may either trap or produce an integer result that is within the range of representable integers on that system. Some applications, particularly in embedded systems, are better handled by producing a verifiably in-range result because it allows the computation to proceed, thereby avoiding a denial-of-service attack. However, when continuing to produce an integer result in the face of overflow, the question of what integer result to return to the user must be considered.

The saturation and modwrap algorithms and the technique of restricted-range usage produce integer results that are always within a defined range. This range is between the integer values `MIN` and `MAX` (inclusive), where `MIN` and `MAX` are two representable integers with `MIN < MAX`.

For saturation semantics, assume that the mathematical result of the computation is `result`. The value actually returned to the user is shown in Table 2.

Table 2: Saturation Semantics

Range of mathematical result	**Result returned**
`MAX < result`	`MAX`
`MIN <= result <= MAX`	`result`
`result < MIN`	`MIN`

In the C standard, signed integer overflow produces undefined behavior, meaning that any behavior is permitted. Consequently, producing a saturated `MAX` or `MIN` result is permissible. Providing saturation semantics for unsigned integers would require a change in the standard. For both signed and unsigned integers, there is currently no way of *requiring* a saturated result. If C1X defined a new standard pragma such as `_Pragma(STDC SAT)`, saturation semantics could be provided without impacting existing code.

Although saturation semantics may be suitable for some applications, it is not always appropriate in security-critical code where abnormal integer values may indicate an attack.

3.4 Overflow Detection

C99 provides the `<fenv.h>` header to support the floating-point exception status flags and directed-rounding control modes required by IEC 60559 and other similar floating-point state information. This support includes the ability to determine which floating-point exception flags are set.

It is ironic that floating point has a set of fully developed methods for monitoring and reporting exceptional conditions, even though the population using those methods is orders of magnitude smaller than the population that needs correctly represented integers. On the other hand, perhaps C's long gestation period for addressing the correct-representation problem will lead to a system that is superior to the other languages that tackled the problem decades ago (such as Pascal and Ada).

A potential solution to handling integer exceptions in C is to provide an inquiry function (just as C provides for floating point) that interrogates status flags that are being maintained by the (compiler-specific) assembler code that performs the various integer operations. If the inquiry function is called after an integral operation and returns a "no overflow" status, the value is reliably represented correctly.

At the level of assembler code, the cost of detecting overflow is zero or nearly zero. Many architectures do not even have an instruction for "add two numbers but do NOT set the overflow or carry bit;"[3] the detection occurs for free whether it is desired or not. But it is only the specific compiler code generator that knows what to do with those status flags.

These inquiry functions may be defined, for example, by translating the `<fenv.h>` header into an equivalent `<ienv.h>` header that provides access to the integer exception environment. This header would support the integer exception status flags and other similar integer exception state information.

However, anything that can be performed by an `<ienv.h>` interface could be performed better by the compiler. For example, the compiler may choose a single, cumulative integer exception flag in some cases and one flag per variable in others, depending on what is most efficient in terms of speed and storage for the particular expressions involved. Additionally, the concept of a runtime-constraint handler did not exist until the publication of ISO/IEC TR 24731-1 [ISO 2006].

---

[3] However, the load effective address (LEA) instruction in Intel architectures is commonly used for integer addition and does *not* set status flags.

Consequently, when designing `<fenv.h>`, the C standards committee defined an interface that put the entire burden on the programmer.

Floating-point code is different from integer code in that it includes concepts such as rounding mode, which need not be considered for integers. Additionally, floating point has a specific value, NaN (Not a Number), which indicates that an unrepresentable value was generated by an expression. Sometimes floating-point programmers want to terminate a computation when a NaN is generated; at other times, they want to print out the NaN because its existence conveys valuable information (and there might be one NaN in the middle of an array being printed out, with the rest of the values being valid results). Because of the combination of NaNs and the lack of runtime-constraint handlers, the programmer needed to be given more control.

In general, there is no NaI (Not an Integer) value, so there is no requirement to preserve such a value to allow it to be printed out. Therefore, the programmer does not need fine control over whether an integer runtime-constraint handler gets called after each operation. Without this requirement, it is preferable to keep the code simple and let the compiler do the work, which it can generally do more reliably and efficiently than individual application programmers.

3.5 Runtime Integer Checking (RICH)

Brumley and colleagues have developed a static program transformation tool, called RICH, that takes as input any C program and outputs object code that monitors its own execution to detect integer overflows and other bugs [Brumley 2007]. Despite the ubiquity of integer operations, the runtime performance penalty of RICH is low, averaging less than 5%. RICH implements the checks in two phases. At compile time, RICH instruments the target program with runtime checks of all unsafe integer operations. At runtime, the inserted instrumentation checks each integer operation. When a check detects an integer error, it generates a warning and optionally terminates the program.

3.6 Clang Implementation

David Chisnall implemented the AIR integer model for Clang using the LLVM overflow-checked operations.[4] The current implementation checks the integer overflow flag after each `+`, `-` or `*` integer operation and calls a handler function on overflow.

In the overflow handler, the operation arguments are promoted to the `long long` type via sign extension; the op indicates whether it was signed/unsigned addition, subtraction, or multiplication; and the width indicates the expected width of the result. GCC's `-ftrapv` can be replicated by having it unconditionally call `abort()`. Alternatively, overflow can be handled by calling a registered handler function from a stack or by promoting it to some kind of boxed value. If this function returns, its return value is truncated and used in place of the operation's result.

The Clang implementation simply checks the flag immediately after any signed integer operation and jumps to a handler function if overflow occurred. Conditional jumps on overflow are cheap because the branch predictor will almost always guess correctly. By allowing a custom handler function, rather than aborting, Clang allows for calling `longjmp()` or some unwind library functions in cases where overflow occurred. This works well with the optimizer, which can eliminate the test for cases where the value can be proven to be in-range.

---

[4] http://article.gmane.org/gmane.comp.compilers.clang.devel/4469

3.7 GCC no-undefined-overflow

Richard Guenther has proposed a new no-undefined-overflow branch for GCC. Its goal is to make overflow behavior explicit per operation and to eliminate all constructs in the GIMPLE intermediate language (IL) that invoke undefined behavior. To support languages that have undefined semantics on overflowing operations such as C and C++, new unary and binary operators that implicitly encode value-range information about their operands are added to the middle-end, noting that the operation does not overflow. These does-not-overflow operators transform the undefined behavior into a valid assumption making the GIMPLE IL fully defined. Consequently, the front-end and value-range analysis must determine if operations overflow and generate the appropriate IL. Instructions such as `NEGATE_EXPR`, `PLUS_EXPR`, `MINUS_EXPR`, `MULT_EXPR`, and `POINTER_PLUS_EXPR` would have wrapping, no-overflow, and trapping variants.

The trapping variants are indicated by a `V` for overflow (e.g., `PLUSV_EXPR` is the trapping variant for `PLUS_EXPR`) and by `NV` for no overflow (e.g., `PLUSNV_EXPR`). The no-overflow variant also wraps if it overflows so that existing code continues to function.

The GCC no-undefined-overflow branch, when implemented, should greatly facilitate the implementation of the AIR integer model within GCC.

3.8 Testing Methods

The majority of vulnerabilities resulting from integer exceptions manifest themselves as buffer overflows while manipulating null-terminated byte strings in C and C++. Yu, Bultan, and Ibarra proposed an automata-based composite, symbolic verification technique that combines string analysis with size analysis that focuses on statically identifying all possible lengths of a string expression at a program point to eliminate buffer overflow errors [Yu 2009]. This technique obviates the need for runtime checks, which is an advantage if the time to perform the checking can be favorably amortized over the expected number of runtime invocations. Runtime property checking (as implemented by AIR integers) checks whether a program execution satisfies a property. Active property checking extends runtime checking by determining if the property is satisfied by all program executions that follow the same program path [Godefroid 2008].

4 Performance & Efficacy Study

A proof-of-concept modification to the GCC compiler Version 4.5.0 was developed for IA-32 processors to study the performance and efficacy of the AIR integer model.[5] The AIR integer model is enabled by the `-fanalyzable` compile-time option. Generated executables automatically invoke a runtime-constraint handler when an integral operation fails to produce an as-if infinitely ranged value.

To diagnose integer overflow on an IA-32 processor, we must know whether the arguments are signed or unsigned so that the appropriate flag (carry or overflow) can be checked. The overflow flag indicates that overflow has occurred for signed operations, while the carry flag can be safely ignored. For unsigned computations, the opposite is true. Unfortunately, GCC's last internal representation, the register transfer language (RTL), has no way of storing the signedness of arguments to operations. Doing so requires inserting a flag into the RTL data structure, the `rtx`, which carries signedness information to the GCC back end, where translation to assembly code is performed. Upon translation, the proper signedness information is available to produce the correct RTL pattern.

Conditional jumps are added to RTL patterns containing arithmetic operations to invoke a runtime-constraint handler in the event that signed overflow or unsigned wrapping occurred, as shown in the following example:

```
   // arithmetic
jn[co] .LANALYZEXXX
   call constraint_handler

.LANALYZEXXX
   // code after arithmetic
```

Overflow checks were not added following signed division because these operations result in a division error on IA-32 and generate an interrupt on vector 0.

4.1 Performance Study

The performance of the prototype was assessed using the industry standard SPECCPU2006 benchmarks, which provide a meaningful and unbiased metric. Because the goal of this project is analyzable integer behavior, we only ran the SPECINT2006 portion of the benchmark. We compiled SPECINT2006 using a reference (unmodified) GCC compiler and a GCC compiler modified to implement branch insertion. Then we ran the two binaries and used the ratio of their runtime to a known baseline to compute a performance ratio.

Higher numbers for the control and analyzable rations in Table 3 indicate better performance.

---

[5] The prototype is available for download at http://www.cert.org/secure-coding/integralsecurity.html.

Table 3: SPECINT2006 Macro-Benchmark Runs

Optimization Level	Control Ratio	Analyzable Ratio	% Slowdown
`-O0`	4.92	4.60	6.96
`-O1`	7.21	6.77	6.50
`-O2`	7.38	6.99	5.58

Because the benchmarks used in SPECINT2006 are not designed for analyzable code, the prototype was modified slightly so that the analysis is performed but programs do not abort in the case of an overflow. The new snippet has a `nop` instruction instead of a call to a runtime-constraint handler:

```
   jn[co] .LANALYZEXXX
   nop
.LANALYZEXXX
```

This modification prevents runtime-constraint handlers from being invoked in the case of overflow and wrapping behavior in the benchmarks.[6]

Code insertions occur after all optimizations are performed by GCC, so the observed slowdown is not caused by disrupted optimizations. Instead, the slowdown is due entirely to the cost of the conditional tests after each arithmetic operation.

Runtime performance could be further improved by eliminating unnecessary tests in cases where value-range analysis can prove that overflow or wrapping is not possible. This technique can be implemented as analyzer advice, where a front-end analyzer provides advice to a back-end compiler. For each input source file, our prototype accepts an analyzer advice file containing either a white list of operations that do not require testing or a black list of operations that do require it. We did not make use of this capability when generating our results because we wanted to establish a baseline prior to introducing any optimizations.

4.2 Efficacy Study

For our efficacy study, the JasPer image processing library and the FFmpeg audio/video processing library were instrumented using our prototype and fuzz tested using zzuf.[7]

JasPer includes a software-based implementation of the codec specified in the JPEG-2000 Part-1 standard ISO/IEC 15444-1 and is written in the C programming language [Adams 2006]. The JasPer software has been included in the JPEG-2000 Part-5 standard ISO/IEC 15444-5 as an official reference implementation of the JPEG-2000 Part-1 codec. This software has also been incorporated into numerous other software projects (some commercial and some non-commercial). Some projects known to use JasPer include K Desktop Environment (as of Version 3.2), Kopete, Ghostscript, ImageMagick, Netpbm (as of Release 10.12), and Checkmark.

---

[6] Ideally, a call instruction would be used because the `nop` instruction is shorter, resulting in better code density.

[7] http://caca.zoy.org/wiki/zzuf

FFmpeg is a multimedia library that supports encoding and decoding a wide range of video and audio formats that are supported by a range of container formats.[8] FFmpeg has been included in a variety of software projects including MPlayer, VLC, Google Chrome, HandBrake, and xine.[9]

These libraries have been included in many commercial and non-commercial applications that have been widely deployed and used. As a result, any vulnerabilities in these libraries are quite severe because they can lead to widespread compromises.

The zzuf tool mangles input to an application while observing the application's behavior. The traditional purpose for fuzz testing is to find test cases that cause an application to crash. However, we used fuzzing to exercise the target application's code paths. Because the applications were instrumented using AIR integers, we were able to observe integer constraint violations as they happened.

Fuzzing with zzuf starts with a *seed* file and randomly mutating bits of the file within a specified percentage range. For our test, we used the range of 0.001% to 1%. Each iteration of the fuzzing run opens the modified seed file, while logging `stderr` output to capture AIR constraint violations. JasPer was fuzzed for 17.5 hours, resulting in execution of the application 1,802,614 times. FFmpeg was fuzzed for 18.2 hours, resulting in execution of the application 978,060 times.

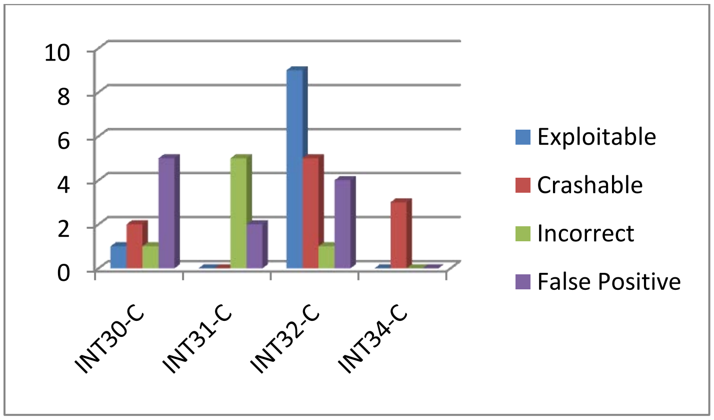

Figure 1: JasPer Defects

The JPEG-2000 decoding capabilities of JasPer were targeted in the fuzzing run. Code coverage details were obtained by using GCC gcov.[10] 74.4% of the code in `jpc_dec.c` , which contains code for decoding JPEG-2000 streams, has been executed through the use of fuzzing.

---

[8] http://ffmpeg.org/general.html#SEC3

[9] http://ffmpeg.org/projects.html

[10] http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

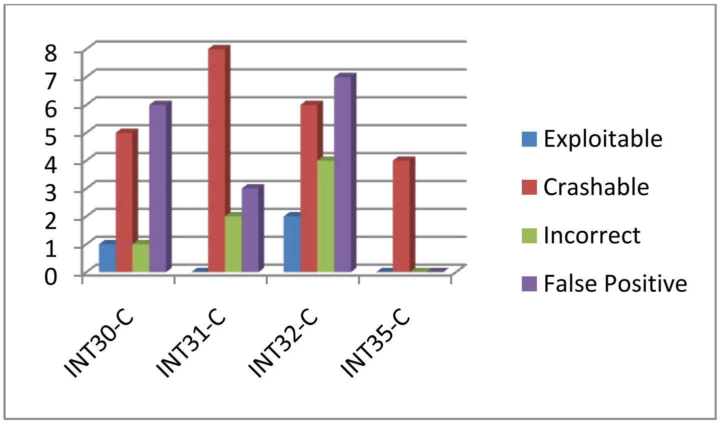

Figure 2: FFmpeg Defects

For the FFmpeg fuzzing run, we targeted Ogg Theora video decoding. Because of the large number of formats supported by FFmpeg, only 7.3% of the entire FFmpeg codebase was covered during the fuzzing run. However, 93.7% of the code in `vp3.c`, which is used for decoding Theora video, has been executed.

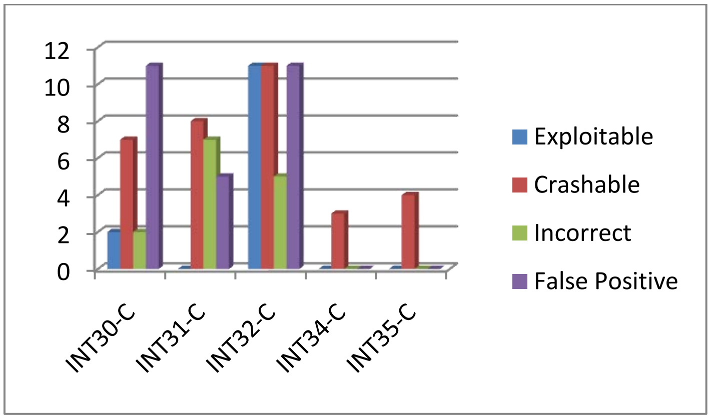

Figure 3: Combined Defects

Figures 1, 2, and 3 show the defects discovered in JasPer and FFmpeg organized by severity and by the *CERT Secure Coding Standard* rule that was violated.

Violations of the following CERT C Secure Coding Standard rules were discovered:

- INT30-C. Ensure that unsigned integer operations do not wrap
- INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data
- INT32-C. Ensure that operations on signed integers do not result in overflow
- INT34-C. Do not shift a negative number of bits or more bits than exist in the operand
- INT35-C. Evaluate integer expressions in a larger size before comparing or assigning to that size

Each runtime constraint was classified as *exploitable*, *crashable*, *incorrect*, or a *false positive*.

Exploitable defects are those that are believed likely to result in an attacker being able to execute arbitrary code.

Crashable defects are those that result in a program crash but whose overall security impact otherwise appears limited to a denial-of-service condition.

Incorrect defects result in incorrect program output or data corruption, but there is no possibility of crashing or exploiting the program.

False positives are traps for overflows or truncations that are not errors because they are harmless for that particular implementation. Technically, these are still defects and may represent undefined behavior or rely on non-portable behaviors. For example, a left shift may be used to extract ASCII character data packed into an `int` or `long`. While this is undefined behavior and a violation of rule INT13-C: “Use bitwise operators only on unsigned operands” in the CERT C Secure Coding Standard [Seacord 2008], it may not constitute a defect for a given implementation.

Instrumented fuzz testing discovered 10 of a known 12 vulnerabilities in JasPer and had no code coverage for the other 2. For comparison, the Splint[11] static analysis tool identified those 2. Of the 10 vulnerabilities discovered through fuzzing, Splint missed 4 and identified 6 but not for the reasons they are actually vulnerable. This is not surprising given that Splint issued 468 warnings for 2000 lines of code.

Another significant difference between the static analysis and AIR runtime strategies lies in the aspect of code coverage. With static tools, the entire codebase is available for analysis. However, with the AIR library, constraint violations are only reported if a code path is taken during program execution and the input data caused a constraint violation to occur. For example, 83 runtime-constraint violations were reported in the `jpc_dec.c` file, while 0 violations were reported in `jpc_enc.c`. That was because the fuzzing run did not perform any JPEG-2000 code stream encoding, and therefore the code in `jpc_enc.c` was never executed.

An example of an exploitable vulnerability by fuzz testing the AIR-integer-instrumented JasPer library occurs in `jas_image_cmpt_create()` where `size` can easily overflow:

```
303: long size;
321: size = cmpt->width_ * cmpt->height_ * cmpt->cps_;
322: cmpt->stream_ = (inmem) ?
       jas_stream_memopen(0, size) :
       cmpt->jas_stream_tmpfile();
```

---

[11] http://www.splint.org/

In `jas_stream_memopen()`, a `bufsize` less than or equal to 0 is meaningful and indicates that a buffer has been allocated internally and is growable:

```
jas_stream_t *jas_stream_memopen(char *buf, int bufsize)

  if (bufsize <= 0) {
    obj->bufsize_ = 1024
    obj->growable_ = 1;
  } else {
    obj->bufsize_ = bufsize;
    obj->growable_ = 0;
  }
```

When `size` overflows in `jas_image_cmpt_create()`, it becomes negative, tricking `jas_stream_memopen()` into thinking it should be allocated internally and be growable.

This problem is diagnosed as follows, identifying both signed integer overflows on line 321 in violation of INT32-C:

```
jas_image_cmpt_create
src/libjasper/base/jas_image.c:321
0x804c8d3 Signed integer overflow in multiplication
0x804c8e3 Signed integer overflow in multiplication
```

An exploitable vulnerability discovered by fuzz testing the AIR-integer-instrumented FFmpeg libraries occurs in `vorbis_residue_decode_internal()` where the variable `n_to_read` can easily overflow:

```
uint_fast16_t n_to_read=vr->end-vr->begin;
uint_fast16_t pnts_to_read=n_to_read/vr->partition_size;
uint_fast8_t classifs[pnts_to_read*vc->audio_channels];
```

The `classifs` array is allocated as a very large array and filled with data that will not be valid:

```
for(i=0;i<c_p_c;++i) {
  uint_fast_t temp2;

  temp2=(((uint_fast64_t)temp) * inverse_class)>>32;
  if (partition_count+c_p_c-1-i < ptns_to_read) {
    classifs[j_times_ptns_to_read+partition_count+c_p_c-1-i]=
            temp-temp2*vr->classifications;
  }
  temp=temp2;
}
```

The data from this `classifs` is then used as an array index later in the function. This array index could be out of bounds:

```
uint_fast8_t vqclass=classifs[j_times_ptns_to_read+partition_count];
int_fast16_t vqbook=vr->books[vqclass][pass];
```

This problem is diagnosed as follows, identifying the unsigned wrapping that occurs on line 1216 in violation of rule INT30-C:

```
vorbis_residue_decode_internal
libavcodec/vorbis_dec.c:1216
0x8745f5c Unsigned integer overflow in subtraction
```

Two false positives in JasPer both involved signed left shift. The first false positive occurs in `src/libjasper/bmp/bmp_dec.c:443` in function `bmp_getint32()`:

```
434: static int bmp_getint32(jas_stream_t *in, int_fast32_t *val)
435: {
436:   int n;
437:   uint_fast32_t v;
438:   int c;
439:   for (n = 4, v = 0;;) {
440:     if ((c = jas_stream_getc(in)) == EOF) {
441:       return -1;
442:     }
443:     v |= (c << 24);
444:     if (--n <= 0) {
445:       break;
446:     }
447:     v >>= 8;
448:   }
449:   if (val) {
450:     *val = v;
451:   }
452:   return 0;
453: }
```

The `jas_stream_getc()` function has similar semantics to `getc()`, meaning it returns `EOF` or an `unsigned char` cast to an `int`. Consequently, the left shift 24 of `c` is safe, provided the platform is non-trapping, although it causes a signed overflow and undefined behavior.

The false positive can be eliminated by casting `c` to an unsigned integer type as follows:

```
v |= ((unsigned int)c << 24);
```

While this is a false positive for vulnerability, it is undefined behavior and is also a violation of rule INT13-C: “Use bitwise operators only on unsigned operands” in the CERT C Secure Coding Standard” [Seacord 2008].

In FFmpeg, harmless unsigned wrapping caused a false positive in `libavformat/utils.c:2322` in the `av_close_input_stream()` function:

```
2322: for(i=s->nb_programs-1; i>=0; i--) {
2323: #if LIBAVFORMAT_VERSION_INT < (53<<16)
2324:   av_freep(&s->programs[i]->provider_name);
2325:   av_freep(&s->programs[i]->name);
2326: #endif
2327:   av_metadata_free(&s->programs[i]->metadata);
2328:   av_freep(&s->programs[i]->stream_index);
2329:   av_freep(&s->programs[i]);
2330: }
```

The false positive occurs when `s->nb_programs == 0` because `s->nb_programs` is defined as an `unsigned int`. Because `i` is declared as an `int`, this unsigned wrapping is perfectly safe.

If desired, the false positive can be eliminated by avoiding the loop when `s->nb_programs == 0` as follows:

```
if (s->nb_programs > 0) {
  for(i=s->nb_programs-1; i>=0; i--) {
    /* ... */
  }
}
```

While this is a false positive for vulnerability, it is a violation of rule INT30-C. "Ensure that unsigned integer operations do not wrap" in the CERT C Secure Coding Standard [Seacord 2008].

5 Conclusions

The AIR integer model produces either a value that is equivalent to a value that would have been obtained using infinitely ranged integers or a runtime-constraint violation. AIR integers can be used for dynamic analysis or as a runtime protection scheme. At the `-O2` optimization level, our compiler prototype showed only a 5.58% slowdown when running the SPECINT2006 macro-benchmark. Although that percentage represents the worst-case performance for AIR integers (because no optimizations were performed), it is still low enough for typical applications to enable this feature in deployed systems. AIR integers have also been proven effective in discovering vulnerabilities, crashes, and other defects in the JasPer image processing library and the FFmpeg audio/video processing library during testing with dumb (mutation) fuzzing.

Bibliography

[Adams 2006]
Adams, Michael D. *JasPer Software Reference Manual (Version 1.900.0)*. December 2006. http://www.ece.uvic.ca/~mdadams/jasper/jasper.pdf.

[Brumley 2007]
Brumley, David; Chiueh, Tzi-cker; Johnson, Robert; Lin, Huijia; & Song, Dawn. "RICH: Automatically Protecting Against Integer-Based Vulnerabilities." *Proceedings of NDSS*, San Diego, CA, February 2007. Reston, VA: ISOC, 2007. http://www.cs.berkeley.edu/~dawnsong/papers/efficient_detection_integer-based_attacks.pdf

[Christy 2007]
Christy, Steve & Martin, Robert A. *Vulnerability Type Distributions in CVE*, Version: 1.1. May 22, 2007. http://cve.mitre.org/docs/vuln-trends/vuln-trends.pdf

[Godefroid 2008]
Godefroid, P.; Levin, M. Y.; & Molnar, D. A. "Active Property Checking," 207-216. *Proceedings of the 8th ACM international Conference on Embedded Software* (EMSOFT '08). Atlanta, GA, October 2008. ACM, NY.

[Gosling 2005]
Gosling, James; Joy, Bill; Steele, Guy; & Bracha, Gilad. *Java Language Specification, Third Edition*. Prentice Hall, 2005.

[Hedquist 2009]
Hedquist, Barry. *International Standards Organization (ISO)/JTC1/SC22/WG14 and InterNational Committee for Information Technology Standards (INCITS) PL22.11 March/April Meeting. Minutes*. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1421.pdf (2009).

[Heffley 2004]
Heffley, J. & Meunier, P. "Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security?" *Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9 - Volume 9* (January 05 - 08, 2004), ISBN:0-7695-2056-1. IEEE Computer Society, 2004.

[Intel 2004]
Intel Corporation. *The IA-32 Intel Architecture Software Developer's Manual*. Intel Corporation, 2004.

[ISO 2010]
International Standards Organization. *Programming Languages—C++, Final Committee Draft,* ISO/IEC JTC1 SC22 WG21 N3092. International Standards Organization, March 2010.

ISO 2006]
International Standards Organization. *ISO/IEC TR 24731. Extensions to the C Library, — Part I: Bounds-Checking Interfaces*. International Standards Organization, April 2006.

[Jones 2009]
Jones, Larry. *WG14 N1401 Committee Draft ISO/IEC 9899:201x*. International Standards Organization, November 24, 2009.

[Lions 1996]
Lions, J. L. *ARIANE 5 Flight 501 Failure Report*. European Space Agency (ESA) & National Center for Space Study (CNES) Inquiry Board, July 1996.

[Plum 2005]
Plum, Thomas & Keaton, David M. "Eliminating Buffer Overflows, Using the Compiler or a Standalone Tool," 75-81. *Proceedings of the Workshop on Software Security Assurance Tools, Techniques, and Metrics.* Long Beach, CA, November 2005. U.S. National Institute of Standards and Technology (NIST), 2005.

[Plum 2009]
Plum, Thomas & Seacord, Robert C. *ISO/IEC JTC 1/SC 22/WG14/N1350 – Analyzability*. International Standards Organization, February 2009. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1350.htm.

[Saltzer 1975]
Saltzer, Jerome H. & Schroeder, Michael D. "The Protection of Information in Computer Systems," 1278-1308. *Proceedings of the IEEE 63*, 9 (September 1975).

[Seacord 2005]
Seacord, R. C. *Secure Coding in C and C++ (SEI Series in Software Engineering)*. Addison-Wesley Professional, 2005.

[Seacord 2008]
Seacord, R. *The CERT C Secure Coding Standard*. Addison-Wesley Professional, 2008.

[Solar Designer 2000]
Solar Designer. *JPEG COM Marker Processing Vulnerability in Netscape Browsers*. OpenWall Project, July 2000. http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt.

[Yu 2009]
Yu, F.; Bultan, T.; & Ibarra, O. H. "Symbolic String Verification: Combining String Analysis and Size Analysis," 322-336. *Proceedings of the 15th international Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*. York, UK, March 2009. Lecture Notes in Computer Science, Volume 5505. Springer-Verlag, 2009.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. **AGENCY USE ONLY**
(Leave Blank)

2. **REPORT DATE**
April 2010

3. **REPORT TYPE AND DATES COVERED**
Final

4. **TITLE AND SUBTITLE**
As-If Infinitely Ranged Integer Model, Second Edition

5. **FUNDING NUMBERS**
FA8721-05-C-0003

6. **AUTHOR(S)**
Roger Dannenberg, Will Dormann, David Keaton, Thomas Plum, Robert C. Seacord, David Svoboda, Alex Volkovitsky, Timothy Wilson

7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. **PERFORMING ORGANIZATION REPORT NUMBER**
CMU/SEI-2010-TN-008

9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
HQ ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2116

10. **SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. **SUPPLEMENTARY NOTES**

12A **DISTRIBUTION/AVAILABILITY STATEMENT**
Unclassified/Unlimited, DTIC, NTIS

12B **DISTRIBUTION CODE**

13. **ABSTRACT (MAXIMUM 200 WORDS)**

Integers represent a growing and underestimated source of vulnerabilities in C and C++ programs. This report presents the as-if infinitely ranged (AIR) integer model that provides a largely automated mechanism for eliminating integer overflow and truncation and other integral exceptional conditions. The AIR integer model either produces a value equivalent to that obtained using infinitely ranged integers or results in a runtime-constraint violation. Instrumented fuzz testing of libraries that have been compiled using a prototype AIR integer compiler has been effective in discovering vulnerabilities in software with low false positive and false negative rates. Furthermore, the runtime overhead of the AIR integer model is low enough for typical applications to enable it in deployed systems for additional runtime protection.

14. **SUBJECT TERMS**
security, standardization, languages, verification, reliability, fuzz testing, software security, integral security, secure coding

15. **NUMBER OF PAGES**
41

16. **PRICE CODE**

17. **SECURITY CLASSIFICATION OF REPORT**
Unclassified

18. **SECURITY CLASSIFICATION OF THIS PAGE**
Unclassified

19. **SECURITY CLASSIFICATION OF ABSTRACT**
Unclassified

20. **LIMITATION OF ABSTRACT**
UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18
298-102